Enhancing Multi-Threaded Legalization Through k-d Tree Circuit Partitioning

Sheiny Fabre, José Luís Güntzel, Laércio Lima Pilla, Renan Netto, Tiago Fontana, Vinicius Livramento

Enhancing Multi-Threaded Legalization Through k-d Tree Circuit Partitioning

Sheiny Fabre[1], José Luís Güntzel[1], Laércio Pilla[2], Renan Netto[1],
Tiago Fontana[1], Vinicius Livramento[1]
[1]Embedded Computing Lab. (ECL), Dept. of Computer Science and Statistics (INE/PPGCC)
Federal University of Santa Catarina (UFSC), Florianópolis, Brazil
[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG – Grenoble, France

Abstract

In the physical synthesis of integrated circuits the legalization step may move all circuit cells to fix overlaps and misalignments. While doing so, it should cause the smallest perturbation possible to the solution found by previous optimization steps to preserve placement quality. Legalization techniques must handle circuits with millions of cells within acceptable runtimes, besides facing other issues such as mixed-cell-height and fence regions. In this work we propose a k-d tree data structure to partition the circuit, thus removing data dependency. Then, legalization is sped up through both input size reduction and parallel execution. As a use case we employed a modified version of the classic legalization algorithm Abacus. Our solution achieved a maximum speedup of 35 times over a sequential version of Abacus for the circuits of the ICCAD2015 CAD contest. It also provided up to 10% reduction on the average cell displacement.

1 Introduction

The physical synthesis of integrated circuits has become a challenging task due to the increasing complexity of design rules and continuous growth in the number of transistors that can be fabricated in a single chip. Circuit fabrication is enabled by physical synthesis, which is responsible for placing and routing the cells' layouts on a 2-D surface [1]. In the physical synthesis flow the global placement step finds an initial solution for the circuit while minimizing interconnection wirelength [2] and a few other constraints such as pin accessibility [3]. However, to be able to find solutions for circuits with millions of gates within acceptable runtime, global placers disregard some layout constraints such as cell overlaps and cell alignment to rows and sites. Therefore, a legalization step must be applied to fix cell overlaps and misalignments. In addition, legalization must preserve the placement quality as much as possible. Legalization may also be invoked several times in the optimization loops within detailed placement techniques such as incremental timing-driven placement [4].

The previously mentioned issues, along with the continuous increase in the number of circuit cells and other contemporary layout constraints (such as mixed-cell-height and rectilinear fence regions), render legalization a relevant research topic for which substantial improvements are still required.

Classic algorithms such as Tetris [5] and Abacus [6] were not conceived to handle those layout constraints. In addition, they do not provide the required performance to handle circuits with millions of cells. Despite of that, a number of state-of-the-art legalizers strongly rely on Abacus [7, 8, 9]. To evaluate the runtime of traditional legalization algorithms we used Abacus to legalize circuits with 760 thousand to 1.93 million of cells. The required runtimes to fully legalize those circuits range from 3.85 to 13.8 minutes, which is not appropriate for iterative optimization loops or even for a single legalization pass.

Legalization algorithms such as Abacus present a quadratic complexity with respect to the number of input elements, which are the circuit cells. That is why a few state-of-the-art legalization techniques partition the circuit and legalize each partition separately [10, 11, 12, 13, 14]. However, since the placement of one cell impacts on its neighbors' placement, there exist dependencies between partitions, which hampers parallelism exploitation. In addition, partitions may present quite different cell densities and, eventually, one partition may become overcrowded and even impossible to legalize.

In this context, we propose an enhanced mechanism for circuit legalization based on the partitioning of the circuits using k-d trees. This mechanism provides a balanced partitioning of the circuits, and enables the parallel execution of legalization algorithms in different partitions while maintaining deterministic solutions. It is also independent of a specific legalization algorithm. Our experimental results with this mechanism implemented on a modified version of Abacus achieved a speedup of 35 over the non partitioned, sequential version on circuits of the ICCAD2015 CAD Contest [15], while also achieving reductions in total and average displacement.

The paper is organized as follows. Section 2 reviews previous techniques and their limitations. Section 3 presents how the k-d tree data structure is built and applied for legalization, and what is important to consider when using this data structure. Section 4 describes the experimental environment and benchmarks. Section 5 presents the results for the proposed technique followed by the conclusions and future works in Section 6.

2 Related work

Several works have been proposed to solve the legalization problem. Hill presents a greedy algorithm called Tetris that legalizes cells one at a time [5]. For each cell, the algorithm finds the row that provides the minimum displacement and places it there. After being placed, a cell cannot be moved by others, which can lead to sub optimal solutions. Abacus [6] is an improvement over this strategy. It is a dynamic-programming algorithm that legalizes the cells within a row. This way, cells can be relocated even after they have been legalized, leading to better solutions than Tetris at a cost of longer runtimes. In order to reduce runtime, Lee et al. [11] propose a partitioning strategy that divides the circuit into bins and calls Abacus inside each bin. If a bin cannot be legalized (due to high cell density), it is merged with its neighbors.

While Tetris and Abacus legalize cells one at a time, the technique proposed by Ren et al. [12] used a diffusion-based method to iteratively spreads cells in order to remove overlaps. Although this strategy in more smooth movements, it does not remove all overlaps and thus requires a final legalization step.

In [10, 13] the legalization problem is modeled as a network flow problem. For this, the circuit is divided into bins, and a bipartite graph is built connecting overflowed and underflowed bins. The network flow problem is solved for this graph in order to identify movements between bins, and then the cells inside each bin are legalized.

Some recent works propose algorithms to handle mixed-cell-height standard cells. For example, Wang et al. [7] adapt Abacus to handle such cells, whereas Hung et al. [14] use a network flow model similar to [10] and [13] while legalizing each bin using an Integer Linear Programming (ILP) model. Meanwhile, Chow et al. [16] propose a new algorithm that enumerates valid insertion points for cells and chooses the one that results in minimum displacement. Chen et al. [3] introduce a new model for the problem as a Linear Complementary Problem by relaxing some of the problem constraints. As a result, they manage to legalize the whole circuit simultaneously, achieving a low perturbation.

Among all of these works, we can notice that some of them partition circuits in order to handle large problems [10, 11, 12, 13, 14]. However, all of them divide the circuit in rectangular bins and assign cells to the closest bin. We believe that the use of a spatial data structure (like k-d trees) can lead to a more balanced circuit partitioning and favoring the parallel legalization of partitions, as discussed in the next section.

3 Using *K*-D trees for circuit partitioning

One way to reduce the execution time of the legalization step lies on parallelizing the used algorithms. Nevertheless, this faces three challenges: 1) physical synthesis algorithms are expected to be deterministic (leading to the same output for the same input), which creates dependencies that reduce the parallelism in the solutions; 2) data dependencies are also present in the legalization phase because the placement of one cell has to take into account the other cells that have been previously legalized; and 3) it requires changes to well-known algorithms, which are error-prone and may not be easily achievable.

3.1 *k*-d trees and their algorithms

A k-d tree is a data structure employed to organize a set of points in a k-dimensional space. It supports insertion, deletion, range searches, and nearest neighbor operations in $\mathcal{O}(n)$ for a k-d tree with n points [17]. A k-d tree can be constructed in $\mathcal{O}(n \log n)$ by recursively choosing the median point in a given dimension as the root and constructing one k-d tree for the set of points before the root and another for the points after the root using a different dimension. Figure 1 illustrates this for a set of ten points in 2 dimensions.

The k-d tree provides a natural way to partition a circuit for four main reasons. First, a median point is selected at each level of the tree and the position of all elements is previously known, thus the resulting k-d tree is always balanced (same number of cells on each sub-tree). This helps keeping the execution times for the partitions similar. Second, no cell may end up crossing a partition line, so the legalization of each partition can be computed independently from the others (but still in a deterministic way). Third, the number of partitions only doubles at each level of a k-d tree, making it easier to control how much to split the circuit. Meanwhile, a quadtree [18] would quadruple the number of partitions for a 2-D domain, making it harder to control the refinement. Fourth and final, if the legalization of a partition is found to not be possible, then we only have to go back to the partition's parent (parent node) and try to legalize it.

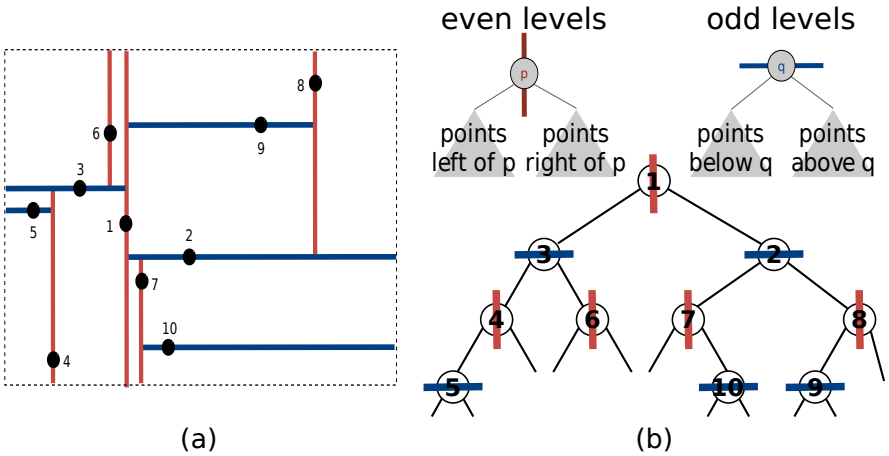

Figure 1: A k-d tree for ten points in a 2-D surface. In (a), the black dots represent the points, and each red and blue line represents a partitioning on the x axis and y axis, respectively. In (b), the k-d tree resulting from (a), in tree representation. Adapted from [17].

We modify the original k-d tree construction algorithm for our needs in two main ways: we include the area information for each sub-tree, and we only build the k-d tree for l levels. The former is used by the legalization algorithm to define each partition boundaries, while the latter serves to limit the number of partitions. This also reduces the construction time from $\mathcal{O}(n \log n)$ to $\mathcal{O}(l\ n)$ (l is a constant and smaller than $\log n$). The modified algorithm for 2-D circuits is presented in Algorithms 1 and 2, while additional details related to the legalization algorithm are presented in Subsection 3.2.

Algorithm 1 takes as input the list of points (bottom left corners of cells) P, the area of the circuit $area$, and the number of levels to create l, and results in the k-d tree partition of the circuit with 2^l partitions. Each node of the tree includes a point, the area of its partition, and left and right child nodes (or a sub-list of points for the partition if a leaf node). Function $first\_axis$, along with function $next\_axis$ in Algorithm 2, provide the partitioning sequence for the k-d tree construction (e.g., x-axis, y-axis, x-axis, etc., for the 2-D domain).

Algorithm 1: *kdtree_build*: Modified k-d tree creation.

Input: List of 2-D points P, circuit area $area$, level limit l.
Output: k-d tree $root\_node$.

```
1 a ← first_axis()                          // initial axis for partition
2 root_node ← kdtree(P, area, a, 1, l)
3 return root_node
```

Algorithm 2 recursively constructs the k-d tree and partitions the circuit. Besides the information given to Algorithm 1, it also takes the current level of the tree c and the current axis for partition a. In Line 3, the median element of P is found and P is partially sorted, meaning that all elements before $P[\frac{|P|}{2}]$ have a smaller position in axis a than $P[\frac{|P|}{2}]$ and the opposite can be said of the elements after $P[\frac{|P|}{2}]$. After the partial sorting of P,it chooses the median point as the root of the sub-tree and splits the points between the ones before and after the median point (P_l and P_r, respectively). It also splits the partition area (in Line 8) and provides this information for the left and right sub-trees recursive calls. This process is repeated over until level l is generated.

The modified algorithm's complexity is in $\mathcal{O}(l\ n)$ for a number of points n ($|P|$) and number of levels l due to all the partial sorting calls required to find the middle points. Nevertheless, this overhead of constructing the k-d tree can be easily overcome by the benefits of partitioning the circuit, e.g., by providing smaller problems for the legalization algorithms and enabling the parallelization of the legalization phase.

3.2 Additional requirements for the legalization problem

In order to employ k-d trees to partition circuits for legalization algorithms, some pre-processing has to be done on the circuit's data. The two steps are as follows:

3.2.1 Overlap removal

All overlaps with fixed cells or macroblocks must be removed before the k-d tree construction, otherwise they may unnecessarily increase the cell displacement as the partitioning may restrict the overlap removal direction,

Algorithm 2: *kdtree*: recursive partitioning function

Input: List of 2-D points P, circuit area $area$, current axis a, current level c, level limit l.
Output: k-d tree $node$.

$node_{area} \leftarrow area$
if $c \leq l$ **then**
  $P \leftarrow nth\_element\_partial\_sort(P, a, \frac{|P|}{2})$
  $m \leftarrow P[\frac{|P|}{2}]$ // median point in P
  $node_{point} \leftarrow m$
  $P_l \leftarrow P[0 : \frac{|P|}{2} - 1]$ // points before m
  $P_r \leftarrow P[\frac{|P|}{2} + 1 : |P| - 1]$ // points after m
  /* divides the area before and after m */
  $area_l, area_r \leftarrow split\_area(m, area, a)$
  /* recursion for left and right children */
  $node_l \leftarrow kdtree(P_l, area_l, next\_axis(a), c + 1, l)$
  $node_r \leftarrow kdtree(P_r, area_r, next\_axis(a), c + 1, l)$
else
  $node_{points} \leftarrow P$ // leaf node
return $node$

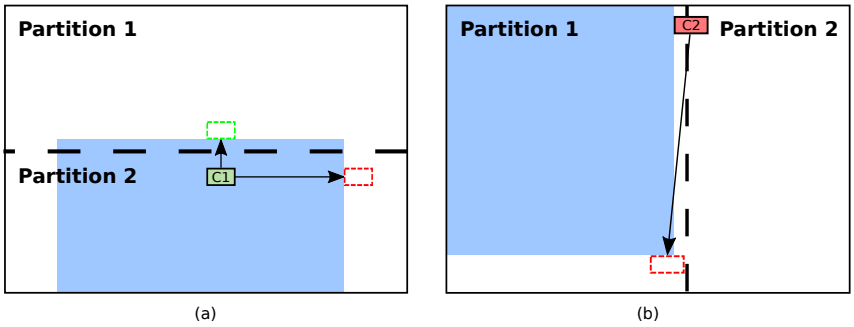

Figure 2: The dashed lines divide the circuit parts into two partitions. The green cell $C1$ overlaps with the blue fixed macroblock, while the red cell $C2$ is tightly placed in partition 1, restricting the cell overlap removal direction during legalization.

and cells cannot move to other partitions after the k-d tree construction. This issue is illustrated in Figure 2 (a), where the green cell $C1$ overlaps with the blue fixed macroblock. In this situation, the minimum overlap removal direction would be vertical, but this movement would not be possible after partitioning the circuit, so the overlap removal would lead to a larger horizontal displacement to the red dashed rectangle.

The overlap removal pre-processing does not solve situations in which a cell is still assigned to a tight place (where it does not fit). This is illustrated in Figure 2 (b), where the red cell $C2$ was moved out of the macroblock but it still does not fit inside Partition 1. As a consequence, this cell will end up being moved to the red dashed rectangle, resulting in a large displacement. Identifying and fixing this kind of situation is left as future work.

3.2.2 Cell alignment correction

A circuit is divided in rows and columns, where the rows are segmented into sites with R_{height} height and S_{width} width. Before legalization, there are usually cells whose positions do not match the alignment of lines and sites of the circuit, as illustrated in Figure 3 (a) for cells $C2$, $C3$ and $C4$. This misalignment has to be fixed (as in Figure 3 (b)) before the k-d tree construction to ensure that the partitions are also aligned and to avoid "dead spaces".

Figure 4 shows what happens when the cells are not aligned before the k-d tree construction (and the ensuing legalization). As the resulting partition boundaries can run across the middle of some lines or sites, the latter become unusable for placing cells, leading to dead spaces in the circuit.

3.3 Legalization using K-D trees

We combine the pre-processing steps from Section 3.2 and the k-d tree algorithms presented in Section 3.1 to enable the legalization of circuits in Algorithm 3. This algorithm takes as input the circuit information

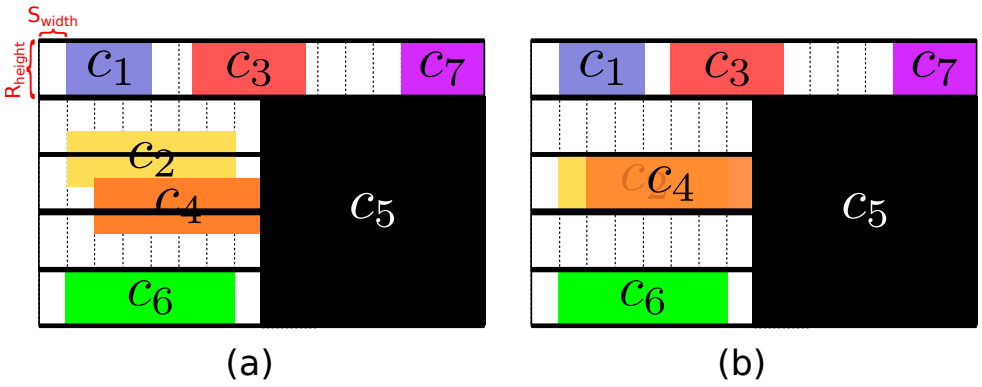

Figure 3: (a): Circuit cells $C2$, $C3$, $C4$ are misaligned to circuit rows and sites, violating the alignment constraint. (b): The cells are now aligned, although an overlap still occurs between cells $C2$ and $C4$.

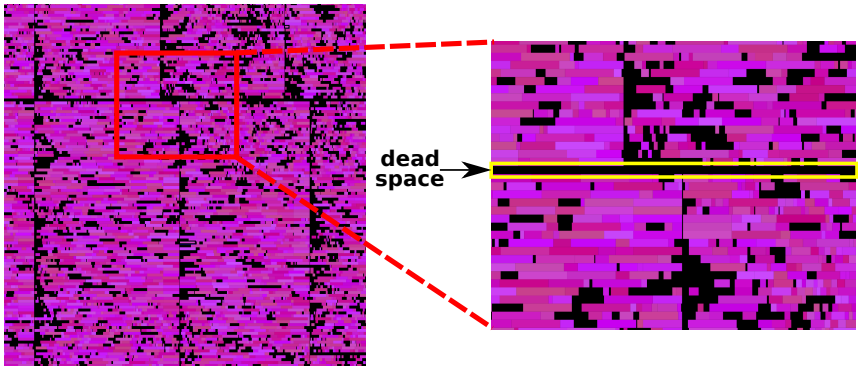

Figure 4: Effect of misaligned cells before partitioning. The rectangles are the circuit cells, and the black rows and sites are free space.

circuit (including its area, macroblocks, and list of cells to legalize), a legalization algorithm L, and the number of k-d tree levels to create l, and returns a list of legalized positions for cells along with the legalization success information (gathered from L). The algorithm removes overlaps and aligns cells (Lines 2 and 3) before constructing the k-d tree. After that, it resets the positions of cells in the k-d tree in order to provide the original position information for the legalization algorithm (Line 5) and then calls the legalization algorithm using the k-d tree as described in Algorithm 4.

Algorithm 3: Circuit legalization using k-d trees.

Input: Circuit information *circuit*, legalization algorithm L, level limit l.
Output: Legalized list of 2-D points C, boolean *success*.

```
   /* Pre-processing                                                    */
1  P ← circuit_cells
2  P ← remove_overlaps_with_marcoblocks(P, circuit)
3  P ← align_cells(P, circuit)
   /* k-d tree construction                                             */
4  node ← kdtree_build(P, circuit_area, l)
5  node ← reset_cell_positions(circuit_cells)
   /* Legalization using the k-d tree                                   */
6  C, success ← kdtree_legalize(L, node, circuit, ∅)
7  return C, success
```

Algorithm 4 uses the k-d tree organization and partitions to guide legalization. If the k-d tree node received is a leaf (verified by function *is_leaf* in Line 1), then the legalization algorithm L is called with the list of points to legalize, the partition area, circuit information (like the positions occupied by macroblocks), and a list of other fixed cells to avoid overlaps (Line 2). If this is not the case, then it will first legalize and fix the node's cell (Line 4) and then it will recursively try to legalize the left and right sub-trees in parallel (Line 6). The algorithm returns a concatenation of the lists of legalized points if the sub-trees were successfully legalized (Line 10), otherwise it gathers all the points in its k-d tree (function *points*) and tries to legalize the current level

partition (Line 12).

Algorithm 4: *kdtree_legalize*: recursive legalization of partitions.

Input: Legalization algorithm L, k-d tree node *node*, circuit information *circuit*, list of fixed parent cell positions F.
Output: Legalized list of 2-D points C, boolean *success*.

```
1  if is_leaf(node) then
       /* Legalizes the cells in the partition                              */
2      C, success ← L(node_points, node_area, circuit, F)
3  else
       /* Legalizes and fixes the root cell                                 */
4      C_root, success_root ← L(node_point, node_area, circuit, F)
5      if success_root then
           /* Legalizes partitions in parallel                              */
6          do in parallel
7              C_l, success_l ← kdtree_legalize(node_l, circuit, F + C_root)
8              C_r, success_r ← kdtree_legalize(node_r, circuit, F + C_root)
           /* Checks if children were legalized                             */
9          if success_l and success_r then
               /* Concatenates children's solutions                         */
10             C, success ← C_l + C_r + C_root, true
11         else
               /* Legalizes the partition at this level instead             */
12             C, success ← L(points(node), node_area, circuit, F)
13     else
14         C, success ← ∅, false                  // Failed to legalize the root cell
15 return C, success
```

Figure 5 illustrates a simple execution of Algorithm 4 for a circuit with eleven cells (A to K) and $l = 2$. The k-d tree with its two levels (shown in Figure 5 (b)) is provided to the algorithm, which will first legalize and fix cell A and then will run in parallel for the left-hand and right-hand sub-trees (nodes with B and C, respectively). For the left-hand sub-tree, the algorithm will legalize and fix cell B and then will run in parallel for its sub-trees. As both are leaves, the legalization algorithm will be called for Partitions 1 and 2, and their results will be combined and returned as the result for the sub-tree with root B. The same will be done for the sub-tree with root C, resulting in the complete legalization of the circuit. Finally, we can note that no cells cross over partitions and all parent cells (white cells in Figure 5 (a)) were legalized before legalizing the partitions, enabling the deterministic and parallel solutions.

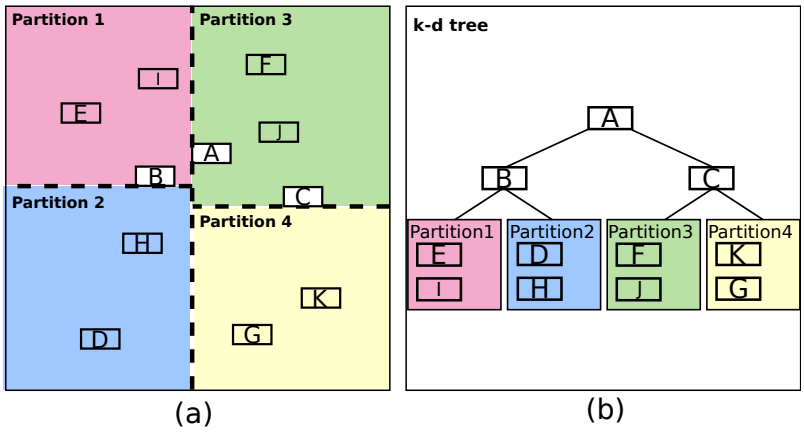

Figure 5: (a): The white cells are the fixed cells and the dashed lines represent the partitions. (b): The k-d tree for the circuit partitioning.

4 Experimental environment

We evaluated our partitioning technique on the circuit benchmarks from the ICCAD2015 CAD contest (problem C: Incremental Timing-driven Placement [15]). It encompasses eight circuits derived from industrial designs, containing from 760k to 1.93 million of cells. The circuits from the more recent ICCAD2017 CAD contest [19] were also considered but not chosen due to their much smaller size (the longest sequential execution took only 26 seconds). Since the ICCAD2015 CAD contest's circuits are already legalized, we applied a vector of random movements on each cell coordinate using a uniform distribution from $-10k$ to $+10k$ dbus. The chosen legalization algorithm was implemented based on a modification of Abacus [6]. As Abacus' complexity is in $\mathcal{O}(n^2)$, the use of k-d trees to partition the circuit in smaller parts should reduce its execution time to about $\frac{n^2}{2^l}$ for n cells and l tree levels. Nevertheless, this change in execution time is not taking into account the overhead of creating and managing the k-d tree (which should increase the total time) and the parallel execution (which should decrease it).

The experiments were executed in a machine with the following specification: one Intel® Core® i5-4460 CPU with 4 cores running at 3.20 GHz, 32GB RAM (4 × 8GB DDR3 at 1600MHz). The code was implemented using the C++ programming language and compiled using gcc version 5.4.0 with the '-O3' optimization flag and all libraries were statically linked (source code available at [20]). We used the OpenMP API [21] to parallelize the code through compiler directives that generate a shared memory parallel program. We executed the parallel version with 4 threads (one per core). During the experiments the threads were bound to cores using hwloc [22].

The execution time results presented in the next section are the arithmetic means of ten executions, whereas the cell displacement results are those obtained by either of the executions, as the solutions are deterministic for each circuit and number of partitions.

5 Results

Figure 6 presents the speedup (ratio of sequential execution times to k-d tree-enabled, parallel legalization times) for the different benchmark circuits with varying numbers of partitions (2^l partitions are generated for a k-d tree with l levels). The parallel execution times also include the necessary pre-processing mentioned in Section 3.2 and described in Algorithm 3. The speedups vary from a factor of 5 for 2 partitions up to about 35 for *superblue16* with 512 partitions (9 levels). A speedup of 5 for 2 partitions comes from reducing the original algorithm time to $\frac{1}{2}n^2$ (i.e., a factor of 2), running the algorithm in parallel for each partition (another factor of 2), and other gains related to locality of reference (due to the reduced problem size) and the line size reduction to be considered by Abacus.

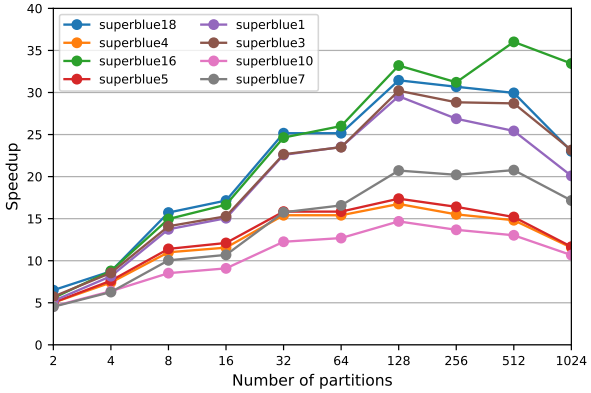

Figure 6: Speedup of the k-d tree-enabled parallel version over the sequential version (bigger is better) for various numbers of partitions 2^l (x axis).

We can also see in Figure 6 that, in general, the speedups start to decrease when we surpass 128 partitions. In this situation, some of the partitions become too small for legalization, so they have to be re-computed in a higher level of the k-d tree. Additionally, the overhead related to managing the k-d tree and the parallel threads can start to become more noticeable. Finally, there is no clear performance trend linked to the number of cells in each circuit. For instance, the two circuits with the smallest speedups in general, *superblue10* and *superblue4*, are the second and penultimate in the number of cells, respectively, among the tested circuits.

Figure 7 displays the improvements in average cell displacement for different numbers of partitions. For a given circuit and a given l, the improvement in average cell displacement was computed as the ratio of average

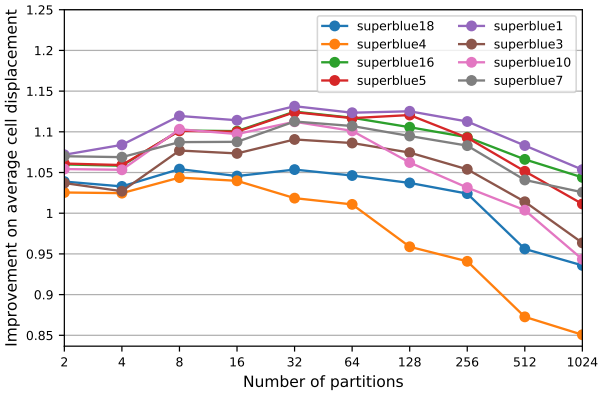

Figure 7: Improvements in average cell displacement (bigger is better and 1 is indifferent). The x axis is the number of partitions 2^l and y axis is the ratio of sequential to k-d tree-enabled parallel version.

cell displacement produced by the sequential version to the average cell displacement from the k-d tree-enabled, parallel legalization version. We can see improvements in the average cell displacement for all circuits when using up to 64 partitions, and for most circuits for all tested numbers of partitions. This is a result of isolating cells to their partitions, which limits how much they can be displaced from their desired positions. Meanwhile, the maximum displacement increases (worsens) as the number of partitions increases, as illustrated in Figure 8. This change is related to situations where the cells do not fit in their desired line or site in the circuit because that would make them cross a partition limit, as previously illustrated in Figure 2 (b). Still, some encouraging results can be seen for *superblue16*, where using 2 or 4 partitions actually improves the maximum displacement.

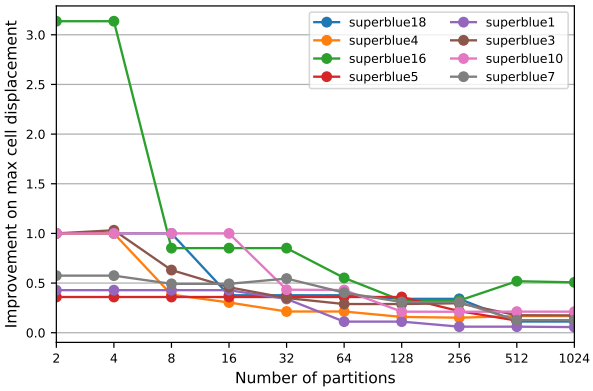

Figure 8: Improvements on maximum displacement (bigger is better and 1 is indifferent). The x axis is the number of partitions 2^l and y axis is the ratio of sequential to k-d tree-enabled parallel version.

6 Conclusions and future works

In this paper, we presented a novel strategy for legalization based on k-d trees to guide the circuit partitioning. Our approach was able to reduce both the execution time of the legalization algorithm and average cell displacement for a set of eight circuits containing from 760k to 1.93 million of cells. These results come from the ability of k-d trees to partition the circuit into independent parts in a balanced manner, enabling the parallel execution of legalization algorithms and providing smaller problem sizes to them.

As future work, we plan to develop an automatic mechanism to determine the ideal number of k-d tree levels to be constructed (instead of using a fixed number of levels). This will require metrics to estimate whether a partition can be legalized or not, or even if it would be profitable to partition even more a part of a circuit. We also plan to evaluate other mechanisms to define where to partition the circuits, and to develop post-processing algorithms to reduce the effects of circuit partitioning on the maximum cell displacement.

References

[1] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI physical design: from graph partitioning to timing closure*. Springer, 2011.

[2] C. Alpert, Z. Li, G.-J. Nam, C. N. Sze, N. Viswanathan, and S. I. Ward, "Placement: Hot or not?" in *Proc. of ISPD*. ACM, 2012, pp. 283–290.

[3] J. Chen, Z. Zhu, W. Zhu, and Y.-W. Chang, "Toward optimal legalization for mixed-cell-height circuit designs," in *DAC 2017*. ACM, 2017, p. 52.

[4] C. Alpert, S. Karandikar, Z. Li *et al.*, "Techniques for fast physical synthesis," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 573–599, 2007.

[5] D. Hill, "Method and system for high speed detailed placement of cells within an integrated circuit design," Apr. 9 2002, uS Patent 6,370,673.

[6] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Abacus: fast legalization of standard cell circuits with minimal movement," 2008.

[7] C.-H. Wang, Y.-Y. Wu, J. Chen, Y.-W. Chang, S.-Y. Kuo, W. Zhu, and G. Fan, "An effective legalization algorithm for mixed-cell-height standard cells," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 450–455.

[8] A. Kennings, N. K. Darav, and L. Behjat, "Detailed placement accounting for technology constraints," in *Very Large Scale Integration (VLSI-SoC), 2014 22nd International Conference on*. IEEE, 2014, pp. 1–6.

[9] S. Popovych, H.-H. Lai, C.-M. Wang, Y.-L. Li, W.-H. Liu, and T.-C. Wang, "Density-aware detailed placement with instant legalization," in *DAC Proceedings*. ACM, 2014, pp. 1–6.

[10] N. Karimpour Darav, I. S. Bustany, A. Kennings, and L. Behjat, "A fast, robust network flow-based standard-cell legalization method for minimizing maximum movement," in *ISPD*. ACM, 2017.

[11] Y.-M. Lee, T.-Y. Wu, and P.-Y. Chiang, "A hierarchical bin-based legalizer for standard-cell designs with minimal disturbance," in *ASP-DAC, 2010 15th Asia and South Pacific*, 2010.

[12] H. Ren, D. Z. Pan, C. J. Alpert, P. G. Villarrubia, and G.-J. Nam, "Diffusion-based placement migration with application on legalization," *IEEE TCAD Integr. Circuits Syst.*, vol. 26, no. 12, pp. 2158–2172, 2007.

[13] U. Brenner, "VLSI legalization with minimum perturbation by iterative augmentation," in *DATE*. IEEE, 2012, pp. 1385–1390.

[14] C.-Y. Hung, P.-Y. Chou, and W.-K. Mak, "Mixed-cell-height standard cell placement legalization," in *ACM GLSVLSI 2017*. ACM, 2017.

[15] M. Kim, J. Hu, J. Li, and N. Viswanathan, "ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite."

[16] W.-K. Chow, C.-W. Pui, and E. F. Young, "Legalization algorithm for multiple-row height standard cell design," in *53th Annual Design Automation Conference 2016*. IEEE, 2016, pp. 1–6.

[17] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[18] H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.

[19] N. K. Darav, I. S. Bustany, A. Kennings, and R. Mamidi, "ICCAD-2017 CAD contest in multi-deck standard cell legalization and benchmarks," in *ICCAD*. IEEE, 2017, pp. 867–871.

[20] "Ophidian: an open source library for physical design research and teaching," https://gitlab.com/sheiny-fabre/ophidian/tree/kdtree.

[21] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, 1998.

[22] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 2010, pp. 180–186.